Protection in the Think exokernel

Christophe Rippert, Jean-Bernard Stefani

Protection in the THINK exokernel

Christophe Rippert*, Jean-Bernard Stefani†
Christophe.Rippert@imag.fr, Jean-Bernard.Stefani@inrialpes.fr

Introduction

In this paper, we present our preliminary ideas concerning the adaptation of security and protection techniques in the THINK exokernel. THINK is our proposition of a distributed adaptable kernel, designed according to the exokernel architecture.

After summing up the main motivations for using the exokernel architecture, we describe the THINK exokernel as it has been implemented on a PowerPC machine. We then present the major protection and security techniques that we plan to adapt to the THINK environment, and give an example of how some of these techniques can be combined with the THINK model to provide fair and protected resource management. Finally, we briefly present the iPAQ Pocket PC to which we plan to port the THINK exokernel and explain our interest in this kind of mobile devices.

1 The Think exokernel

Traditional kernels [1, 2] provide the application programmer with abstractions like virtual memory, processes, or file systems, to ease the development of applications. The programmer can use these abstractions to avoid coping directly with the hardware, which is usually a tedious work. However, monolithic kernels are often considered as bulky, poorly evolutive and slow [3] by kernel developers who have been trying to find alternative solutions for more than 30 years [4].

1.1 Micro-kernels

The micro-kernel architecture [5] has been proposed to improve the portability, the modularity and the evolutivity of standard kernels. A micro-kernel includes the basic abstractions that all standard applications should need, such as a memory manager or inter-process communications. A micro-kernel can be extended with additional abstractions structured as servers outside the micro-kernel itself, that specific applications requiring them can call using IPC. The micro-kernel architecture offers a better control over the hardware resources to the application programmer than monolithic kernels, since it provides only the most basic abstractions and does not force the programmer to use high-level abstractions including functions that he might not need. However, it has been deemed insufficient for applications that require a fine-grained control of the hardware resources, to enjoy high performance or to enforce application specific policies.

1.2 Exokernels

The exokernel architecture [6, 7] is based on the idea that a kernel should not force the programmer to use any abstractions, even the most basic ones. So an exokernel only propose interfaces which give the programmer direct access to the hardware resources, without adding any functionality. High-level abstractions can be provided but their use must remain completely optional to the application programmer. For instance, a programmer who needs a classic scheduler for his application could use

*Université Joseph Fourier
†Institut National de Recherche en Informatique et en Automatique

one provided by the system, but nothing should prevent him to implement and use its own scheduler if he needs to. By providing these optional abstractions as external libraries, the exokernel architecture offers a complete flexibility to the programmer. Moreover, the modularity of the exokernel architecture enables the application programmer to choose exactly which libraries his application needs and to install only these libraries, whereas a monolithic kernel is usually bloated with services that only a few (if any) applications really need.

1.3 The Think exokernel

The THINK architecture is our proposition of a distributed exokernel. As its name implies, THink Is Not a Kernel. It does not provide any abstractions usually proposed by traditional kernels, like a process model or a scheduler for example. Instead, it provides interfaces that export the hardware resources to the applications. It also supplies binding factories to permit communication between the objects that will compose the application. A configuration tool is proposed to help the programmer define which interfaces are needed by a given object, and which interfaces it provides, and to check that all needed interfaces are available at compilation time. Finally, some standard abstractions (memory or a process models for example) are also provided, but their use is completely optional and the application programmer can chose to implement its own abstractions if they suit his application better.

The hardware interfaces provided by THINK are completely machine-dependent. They do not extend the hardware functionalities nor do they try to ease the portability of the architecture by providing high-level functions common to several machines. Their aim is simply to give access to hardware resources by wrapping them in software interfaces. For instance, a TrapRegister(id, handler) function is included in the exception-handling interface. Using this function to register an exception handler is much easier than manipulating the exception vector table directly, without adding any functionality to the processor exception model.

In the THINK architecture, software and hardware resources are seen as objects, as defined in the ODP Reference Model [8]. These objects export interfaces that define their behavior to other objects. Each interface has a name in a given naming context, and is linked with others by bindings. A binding is basically a communication channel between objects. Bindings can take many forms, as simple as the association between a variable name and its value in memory, or more complex like a network connection between objects on different machines. Bindings are created by dedicated objects, called binding factories, which basic functionality (i.e. creating a binding between the calling object and an interface identified by its name) can be freely extended to ensure a given behavior. Finally, objects are grouped in domains according to a common property (e.g. security domains, fault domains, ...).

These various concepts are represented as a minimal software framefork, described in the Figure 1 below. We use Java as the interface description language in THINK.

```
interface Top {
}

interface Name {
  NamingContext getDefaultNamingContext();
  String toString();
}
```

```
interface NamingContext {
  Name toName(String name);
  Name export(Top itf);
}

interface BindingFactory {
  Top bind(Name name);
}
```

Figure 1: The core software framework in THINK

The `Top` interface is the greatest element in the THINK type lattice, the common type from which all interfaces derive. The `Name` interface is the common type for all names in THINK. The method `NamingContext getDefaultNamingContext()` returns the current naming context and the method `String toString()` provides a serialized form of the name. The `NamingContext` interface is the

common type for all naming contexts in THINK. Its method `Name toName(String name)` deserializes a name known as a String. The method `Name export(Top itf)` provides a name for a given interface. As a side effect, it also creates a binding between the returned name and the given interface. The `BindingFactory` interface is the common type for all binding factories in THINK. The method `Top bind(Name name)` creates a binding between the calling object and the object those name is given to the method.

The THINK architecture has been implemented by Jean-Philippe Fassino from France Télécom R&D on a PowerPC computer. More details about this experiment, including benchmarks, can be found in [9].

2 Security issues

The exokernel architecture seems to be much less secure that a monolithic kernel. Indeed, allowing the application programmer to directly access the hardware resources opens the way to all sorts of abuses. Thus, some exokernel specific security techniques have been devised, like the secure bindings presented in [6]. We plan to further this work by adapting existing security techniques used in standard operating systems and virtual machines to the exokernel architecture. We give below an example of how to use some of these techniques in the THINK environment to implement a fair scheduler.

2.1 Existing security techniques

Our definition of security threats is the same as in [14]: an application should not be able to access private data belonging to another application without permission, and no application should monopolize the system resources in a way that would compromise the quality of the service offered to other applications.

Much work has been conducted around the isolation of applications (or processes) to prevent unauthorized accesses. Security techniques usually exploit the hardware to implement efficient isolation [15]. However, some processors do not offer these kinds of facilities, and basing a system security on them reduces the portability of this system. So software isolation techniques have been devised to provide machine-independent security [16, 17], without penalizing IPC too heavily.

Apart from preventing unauthorized accesses, processes isolation is also a way to prevent a faulty process from compromising the whole system. Once again, this isolation is often achieved by using hardware facilities, but efficient software-based isolation techniques have also been devised [18].

However, the most challenging security problem is the fair allocation of resources between the various applications running in the system. A process should not be able to use all the memory space or monopolize the processor, even without malign intentions. The main difficulty lies in counting exactly the amount of resources allocated to a process (for example, should each process using an area of shared memory be charged for the whole area or for a fraction of it?). Isolating the processes can be a way to ease the count of resources allocated, though it usually complicate IPC (forbidding shared memory usually complicate data exchange between processes, for example) [19]. Solutions have been proposed to enforce fair resources allocation without compromising IPC [20, 21], though these propositions are centered on the Java environment. A Java interpret could be implemented in THINK as an optional abstraction, but not in the exokernel itself. Moreover, language-based security has been deemed less effective than operating system based protection [22], since the former implies that the language compilation chain must be trusted. Operating system based security has been studied too [23, 24] concerning the fair allocation of resources. Adaptation of security has also been studied in the Gandiva system, which provides configuration support for applications written in C++ [25] and Java [26].

2.2 Example: implementing a fair and protected scheduler in Think

In a standard operating system, the kernel is responsible for resource allocations, so it is easy to implement a security system in it. On the other hand in the exokernel architecture, resources are allocated by applications, not by the kernel, which greatly complicates the implementation of a protection

system to enforce fair allocation of resources for example. We believe that the binding factories can be used to enforce this security, as we shall see in the following example.

The Figure 2 describes the basic architecture for a fair and protected scheduler abstraction in THINK. We suppose that the programmer's application is composed of three processes (this model can of course be extended to any number of processes). Each one is executed in its own security domain (D1, D2, D3). These domains can be defined using software-based fault isolation techniques as presented in [18] for example. The scheduler itself is executed in its own domain. To share the processor between the processes, the scheduler needs to access the system clock, which is considered as a hardware resource in THINK. So the scheduler must register a new trap-handler for the clock interruption, using the TrapRegister function described above. By doing this, the scheduler creates a binding between itself and the trap object located in the exokernel. This binding is represented by the solid line in the Figure 2 (the dashed lines represents the fact that the three processes are scheduled by the scheduler object). So the scheduler is able to start and stop the processes according to its own policy. Obviously, the three processes must not be able to modify the clock trap handler. Hence, the binding factory that creates the binding between the scheduler and the system clock must check that any binding requests for the system clock come from a scheduler object and not a standard process.

This architecture can easily be provided as an optional library. Then the application programmer only needs to implement his own scheduler and instantiate the library with his scheduler and processes. Therefore, this abstraction can easily be adapted to provide fair resource allocation techniques such as those presented in [21] for example. The binding factory responsible for providing bindings to the system clock object can easily be extended to deny bindings with non-scheduler objects, thus providing a secure binding mechanism as presented in [6]. We believe that this protection scheme can be generalized to other resources allocation.

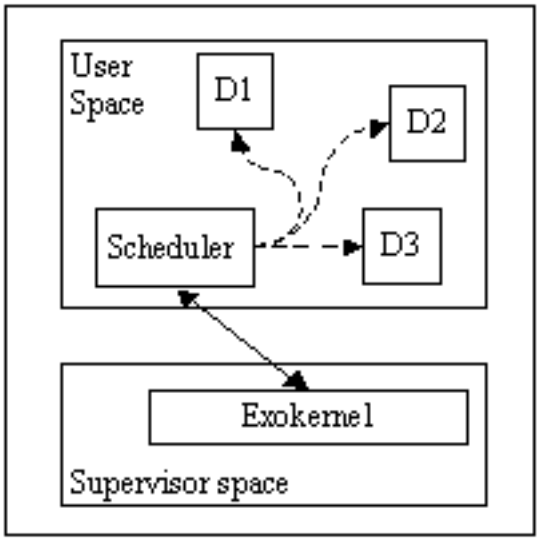

Figure 2: An fair and protected scheduler in THINK

3 Porting Think to a mobile computer

As an application of our work on exokernels and protection, we plan to port THINK and its associated libraries to the iPAQ Pocket PC [10]. This mobile device includes a 206 MHz Intel StrongARM processor [11], 32 megabytes of SDRAM and 16 megabytes of flash memory, which make it powerful enough to run interesting applications. The Windows Pocket PC operating system [12] is initially installed in the flash memory, but a port of Linux has been implemented by volunteers supporting the Open Source Software movement on handheld computers [13]. This will enable us to reuse some

of the existing Linux code (especially the device drivers), which will be easier than programming the libraries from scratch.

Our main interest in porting the THINK architecture to the iPAQ lies in the inherent limitations of this type of mobile devices. Even if the iPAQ is powerful enough for a mobile device (a 200 MHz processor and 32 MB of RAM was a good configuration for a standard PC five years ago), it cannot compete with modern machines with processor speeds approaching the GHz and memory sizes of 256 or 512 MB. Therefore, using an exokernel on this kind of devices can help the application programmer to optimize his applications to use efficiently the available resources. Considering the small amount of memory available (32 MB of RAM can be considered a small amount nowadays), it should not be wasted by including unnecessary services in the kernel, for example. Likewise, the processor should work more on the application code than on the kernel code, and it should never work on unnecessary code. This is especially important since the power consumption is directly linked to the activity of the processor, and power is a precious resource on a battery-powered device like a Pocket PC. We think that using an exokernel on this type of devices can increase its efficiency (i.e. speed up the execution of applications and lengthen the battery life), which we shall try and prove by monitoring the performances of our architecture and comparing it with the same machine running Linux and Windows Pocket PC OS.

Conclusion

We are interested in building a secure and efficient kernel on a mobile computer such as the iPAQ Pocket PC. We chose the exokernel architecture since we consider it to be the best-suited environment to provide the application programmer with a fine-grained control over the hardware resources. Moreover, the modularity of the THINK exokernel permits to restrict the proposed services to those that the applications really need. By using secure bindings to implement the security techniques used in standard operating systems, we plan to provide the same level of security in the exokernel architecture that in a classical monolithic kernel. We believe that the exokernel is a good architecture to build secure, adaptable, and efficient kernels and we hope to contribute to its promotion by porting THINK to different kinds of devices.

References

[1] David A. Rusling. The Linux Kernel. January 1998, `http://www.linuxdoc.org`.

[2] David A. Solomon and Mark Russinovich. Inside Microsoft Windows 2000, Third Edition. Microsoft Press, August 2000.

[3] Dawson R. Engler and M. Frans Kaashoek. Exterminate All Operating System Abstractions. Workshop on Hot Topics in Operating Systems, May 1995.

[4] B. W. Lampson. On reliable and extendable operating systems. *State of the Art Report, Vol.1, Infotech Ltd*, 1971.

[5] Michel Gien. Micro-Kernel Architecture, Key to Modern Operating Systems Design. *Unix Review, Vol. 8, No 11*, November 1990.

[6] Dawson R. Engler, M. Frans Kasshoek and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. *ACM Symposium on Operating Systems Principles*, 1995.

[7] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. *ACM Symposium on Operating Systems Principles*, 1997.

[8] ODP Reference Model. Fundations. ITU-T ISO/IEC Recommendation X.902 - International Standard 10746-2, 1995.

[9] Jean-Philippe Fassino and Jean-Bernard Stefani. Think : un noyau d'infrastructure répartie adaptable. *2ème Conférence Française sur les Systèmes d'Exploitation*, Avril 2001.

[10] iPAQ Pocket PC home page. `http://www.compaq.com`.

[11] Intel StrongARM SA-1110 Microprocessor Developer's Manual. June 2000. `http://developer.intel.com`.

[12] Microsoft Corporation Pocket PC home page. `http://www.microsoft.com`.

[13] Handhelds.org home page. `http://www.handhelds.org`.

[14] Butler W. Lampson. Protection. *ACM Operating Systems Review*, January 1974.

[15] Tzi-cker Chiueh and Ganesh Venkitachalam and Prashant Pradhan. Integrating Segmentation and Paging Protection for Safe Efficient and Transparent Software Extensions. *ACM Symposium on Operating Systems Principles*, 1999.

[16] Dan S. Wallach and Dirk Balfanz and Drew Dean and Edward W. Felten. Extensible Security Architectures for Java. *ACM Symposium on Operating Systems Principles*, 1997.

[17] Jeffret S. Chase and Henry M. Levy and Michael J. Feeley and Edward D. Lazowska. Sharing and Protection in a Single Address Space Operating System. *ACM Transactions on Computer Systems*, May 1994.

[18] Robert Wahbe and Steven Lucco and Thomas E. Anderson and Susan L. Graham. Efficient Software-Based Fault Isolation. *ACM Special Interest Group on OPerating Systems*, 1993.

[19] Godmar Back and Wilson Hsieh. Drawing the Red Line in Java. *IEEE Workshop on Hot Topics in Operating Systems*, March 1999.

[20] Godmar Back and Patrick Tullmann and Leigh Stoller and Wilson C. Hsieh and Jay Lepreau. Java Operating Systems: Design and Implementation. University of Utah, School of Computing, Technical report UUCS-98-015, August 1998.

[21] Godmar Back and Wilson C. Hsieh and Jay Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. *ACM Symposium on Operating Systems Design and Implementation*, October 2000.

[22] Trent Jaeger and Jochen Liedtke and Nayeem Islam. Operating System Protection for Fine-Grained Programs. *USENIX Security Symposium*, January 1998.

[23] David R. Cheriton and Kenneth J. Duda. A Caching Model of Operating System Kernel Functionality. *ACM Symposium on Operating Systems Design and Implementation*, November 1994.

[24] Bryan Ford and Sai Susarla. CPU Inheritance Scheduling. *ACM Symposium on Operating Systems Design and Implementation*, October 1996.

[25] Stuart M. Wheater, Mark C. Little. The Design and Implementation of a Framework for Configurable Software. *IEEE International Conference on Configurable Distributed Systems*, May 1996.

[26] Mark C. Little, Stuart M. Wheater. Building Configurable Applications in Java. *IEEE International Conference on Configurable Distributed Systems*, May 1998.